(2)

AD-A237 144

SSD-TR-90-49

The Language for DENOTE (Denotational Semantics Translation Environment)

Prepared by

J. V. COOK
Computer Science and Technology Subdivision

31 August 1990

Prepared for

SPACE SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Base
P. O. Box 92960
Los Angeles, CA 90009-2960

Engineering Group

THE AEROSPACE CORPORATION
El Segundo, California

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract No. F04701-88-C-00890 with the Space Systems Division, P. O. Box 92960, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by C. A. Sunshine, Principal Director, Computer Science and Technology Subdivision. Mike Pentony, Lt, USAF, was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) Program.

[illegible]

Mike Pentony, Lt, USAF
MOIE Project Officer
SSD/SDEFS

[illegible]

Jonathan M. Emmes, Maj, USAF
MOIE Program Manager
PL/WCO OL-AH

1. INTRODUCTION

This report describes the language accepted by DENOTE (Denotational Semantics Translation Environment). DENOTE is a tool for writing and implementing formal denotational semantics† (Ref. 1) of computer languages. DENOTE provides a language, the DENOTE language (DL), for writing denotational semantic specifications. DENOTE can transform DL specifications into denotational semantic equations and functions represented as plain ASCII text or as formatted text acceptable to the Scribe (Ref. 2) and LaTEX (Ref. 3) text formatters. In addition, DENOTE can generate semantic implementations, by automatically transforming DL specifications into Common Lisp (Ref. 4) code that faithfully implements the specified semantics.

This introduction is followed by four sections, an appendix, and an index. Section 2 gives some background on DENOTE. Section 3 describes the notations used in defining the abstract syntax of programming languages, and presents some restrictions on the types of abstract syntax trees accepted by DENOTE. Section 4 describes the DENOTE !anguage (DL), in which denotational semantic equations and functions are written. Section 5 summarizes the report. The appendix gives a small example that demonstrates the use of DENOTE. The index provides a cross reference to the reserved words of DL. The implementation of DENOTE, and its user interface, are not described in this report.

†A familiarity by readers of this report with the concepts of denotational semantics is assumed.

ABSTRACT

The State Delta Verification System (SDVS) is a proof checker for correctness proofs of properties expressed in the state delta logic. When one proves correctness properties of a computerc program within SDVS, one must first translate the program into the language of the state delta logic. The translation semantics of a computer language can be specified formally by means of denotational semantics. In this report we describe an automated environment for specifying and implementing denotational semantics, called DENOTE (Denotational Semantics Translation Environment). The language accepted by DENOTE is called the DENOTE language (DL). DL is a language in which formal denotational semantic specifications can be written. DL specifications can be transformed by DENOTE into text suitable for input to the Scribe and LaTEX text formatters, as well as into Common Lisp code that implements the specified semantics.

CONTENTS

FIGURES

TABLES

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release: distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-0090(5920-07)-02	5. MONITORING ORGANIZATION REPORT NUMBER(S) SSD-TR-90-49
6a. NAME OF PERFORMING ORGANIZATION The Aerospace Corporation Computer Systems Division; 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Space Systems Division
6c. ADDRESS *(City, State, and ZIP Code)* El Segundo, CA 90245	7b. ADDRESS *(City, State, and ZIP Code)* Los Angeles Air Force Base Los Angeles, CA 90009-2960
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F04701-88-C-0089
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
The Language for DENOTE (Denotational Semantics Translation Environment)

12. PERSONAL AUTHOR(S) Cook, J. V.

13a. TYPE OF REPORT	13b. TIME COVERED FROM ______ TO ______	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
		1990 August 31	29

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer verification
Denotational semantics
Language translation
SDVS

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The State Delta Verification System (SDVS) is a proof checker for correctness proofs of properties expressed in the state delta logic. When one proves correctness properties of a computer program within SDVS, one must first translate the program into the language of the state delta logic. The translation semantics of a computer language can be specified formally by means of denotational semantics. In this report we describe an automated environment for specifying and implementing denotational semantics, called DENOTE (Denotational Semantics Translation Environment). The language accepted by DENOTE is called the DENOTE language (DL). DL is a language in which formal denotational semantic specifications can be written. DL specifications can be transformed by DENOTE into text suitable for input to the Scribe and LaTeX formatters, as well as into Common Lisp code that implements the specified semantics.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* ; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

2. BACKGROUND

The impetus for the design and construction of the DENOTE tool came from certain requirements necessary for the improvement of the State Delta Verification System (SDVS) (Ref. 5). SDVS is a proof checker for proofs of correctness of the properties of programs written in various computer languages. SDVS, when checking such proofs, first translates the computer program into SDVS's internal language, the language of the state delta logic. The correctness of this translation is crucial to the proof process, in that the translator is directly responsible for specifying the semantics of the computer program to be translated.

The first computer language dealt with by SDVS was the hardware description language ISPS (Instruction Set Processor Specification) (Ref. 6). ISPS was used to describe the instruction set architecture of the C/30 computer (Ref. 7) and to describe the Microprogrammable Building Block (Ref. 8), whose microcode was used to emulate the C/30 instruction set. These ISPS descriptions were used in the proof of correctness of the C/30 microcode (Ref. 9), completed in 1986. The ISPS translator used in this proof was implemented in an *ad hoc* manner, in that no formal definition of the translation semantics was given. Only through studying the Lisp code that implements the translator could one gain assurance as to the correctness of the translator.

Initiated in 1988, an ongoing computer verification effort of the SDVS group is proving correctness properties of programs written in increasingly complex subsets of Ada[†] (Ref. 10). The first of these subsets was given the name Core Ada. Its denotational semantics was defined formally (Ref. 11) and was then implemented by handcoding the denotational semantic equations and functions in Common Lisp (Ref. 4). This task was time consuming, tedious, and almost automatic for the programmer, leading to the realization that the entire process could be automated. This realization led to the invention of DENOTE.

DENOTE is currently being used to define state delta semantics for the hardware description languages ISPS and VHDL (Ref. 12), and the programming language Ada.

[†]Ada is a registered trademark of the U. S. Government --- Ada Joint Program Office.

3. ABSTRACT SYNTAX

The denotational semantics of a computer language is typically based on an *abstract* syntax for the language. The *concrete* syntax of a computer language is the syntax used to parse sentences in the language. A parse tree based on the concrete syntax contains a complete derivation of the parsed program, with an internal node for each concrete syntax production that is reduced during parsing. In abstracting the concrete syntax, the resulting parse trees (abstract syntax trees) become smaller and more manageable. The differences between the two syntaxes are best explained via an example.

```
                          expr
                           |
                          sum
                      /    |    \
                factor   addop    sum
                  |        |       |
                 term      +     factor
                  |                |
                 bnum             term
                 /  \              |
          bdigit     bnum         bnum
            |         /\           /\
            1   bdigit   bnum  bdigit  bnum
                  |       |      |      |
                  0     bdigit   1    bdigit
                          |             |
                          1             1

     (expr (sum (factor (term (bnum (bdigit 1)
                                    (bnum (bdigit 0)
                                          (bnum (bdigit 1))))))
                (addop +)
                (sum (factor (term (bnum (bdigit 1)
                                         (bnum (bdigit 1))))))))
```

Figure 3-1: Parse Tree for 101+11 using Concrete Syntax

Consider the context-free language $\mathcal{L}$ generated by the concrete syntax productions listed below.

```
expr    ::= sum
sum     ::= factor addop sum | factor
factor  ::= term mulop factor | term
term    ::= bnum | ( expr )
addop   ::= + | -
mulop   ::= * | /
bnum    ::= bdigit bnum | bdigit
bdigit  ::= 0 | 1
```

Each sentence in this language represents an arbitrary arithmetic term involving the operations of addition, subtraction, multiplication, and division applied to sequences of binary digits. The concrete parse tree for the sentence 101+11 is shown in Figure 3-1. The nonterminal symbols of these concrete syntax productions are expr, sum, factor, term, addop, mulop, bnum, and bdigit; the terminal symbols are (,), +, -, *, /, 0, and 1. The notations ::= and | represent derivation and alternation, respectively. Note that no pseudoterminals[†] are used to define $\mathcal{L}$ (either bnum or bdigit could be pseudoterminals).

```
           plus
            /\
    bdigits   bdigits
       |         |
    (1 0 1)    (1 1)


(plus (bdigits (1 0 1)) (bdigits (1 1)))
```

Figure 3-2: Parse Tree for 101+11 using Abstract Syntax

Productions defining an abstract syntax for $\mathcal{L}$ are shown below.

expr ::= binary-op expr_1 expr_2 | **BDIGITS** bdigit^+

binary-op ::= **PLUS** | **MINUS** | **MULT** | **DIVIDE**

bdigit ::= 0 | 1

In the abstract syntax, expressions are generalized (operator precedence explicit in the concrete syntax no longer requires representation) and production sequences are compressed. The abstract parse tree for the sentence 101+11 is shown in Figure 3-2. The nonterminal symbols of the abstract syntax productions are expr, binary-op, and bdigit; the terminal symbols are **BDIGITS**, **PLUS**, **MINUS**, **MULT**, **DIVIDE**, 0, and 1.

3.1. A Language for Abstract Syntax

Before formally defining the language for representing abstract syntax productions, some notation must be introduced. Itemized below are language notations for terminal symbols, sequences of (arbitrary) symbols, the empty sequence, and a syntactic method for disambiguating multiple occurrences of symbols in a production.

- `'x` denotes a terminal symbol `x`.
- `(x +)` denotes a sequence of one or more `x`.
- `(x *)` denotes a sequence of zero or more `x`.
- `epsilon` denotes the empty derivation.
- `(x n)`, where `n` is a symbol (typically a natural number), may be used to disambiguate

[†]A pseudoterminal denotes a *class* of terminal symbols, such as "identifiers" or "numerals."

multiple occurrences of the (nonterminal or pseudoterminal) symbol `x` in an individual production. The symbol `n` *qualifies* `x`, by distinguishing it from other occurrences of `x` in the production. In a like manner, `(x * n)` may be used to qualify an occurrence of `(x *)` and `(x + n)` to qualify an occurrence of `(x +)`. DENOTE requires qualification in a production when more than one occurrence of a nonterminal or pseudoterminal symbol appears in the production.

For example, consider a syntactic phrase[†] of $\mathcal{L}$, derived from the nonterminal expr, shown below:

PLUS expr_1 expr_2

The DENOTE language (DL) notation for this phrase is

```
('PLUS (expr 1) (expr 2))
```

Without the qualification of `expr`, DENOTE would have no natural way of distinguishing between the two when referencing them from within a semantic equation.

The formal definition of the DENOTE language for representing abstract syntax is given below.

- An *abstract syntax* is a list of *productions*.
- A *production* is a list whose first element is a nonterminal and whose remaining elements are the *alternatives* generated by (derived from) the nonterminal. The symbol *epsilon* denotes the empty derivation. Each list representing a production must have at least two elements, the simplest production being `(nt epsilon)`, denoting a nonterminal `nt` that has only the empty derivation. If the nonterminal defining the production also appears within one of the alternatives, duplicate occurrences of the nonterminal must be qualified for disambiguation.
- An *alternative* is either a terminal symbol, a nonterminal symbol (or sequence thereof), a pseudoterminal symbol (or sequence thereof), or a list of the aforementioned symbols.

A DL representation of the abstract syntax for the example language $\mathcal{L}$ is shown below.

```
((expr (binary-op (expr 1) (expr 2))
       ('bdigits (bdigit +)))
 (binary-op 'plus 'minus 'mult 'divide)
 (bdigit '0 '1))
```

If `bdigit` was defined to be a pseudoterminal of $\mathcal{L}$, the final production would be absent.

3.2. Implementation Requirements for Abstract Syntax Trees

DENOTE requires a particular physical representation of abstract syntax trees. These trees must conform to the following rules:

- An abstract syntax tree contains no nonterminal symbols, only terminal and pseudoterminal symbols and lists thereof. DENOTE is case-insensitive.
- Terminal symbols appear verbatim.
- A pseudoterminal is represented by the appearance of the particular symbol parsed.
- A sequence of symbols (defined by either the $^+$ or * notation) is epresented by a list containing each symbol's representation, or by the empty list if the sequence is empty.

[†] *By "syntactic phrase" we mean some derivation of a nonterminal from the abstract syntax.*

- An alternative that is defined in the abstract syntax by a list of symbols must be represented by a list that contains the ordered representation of each symbol, the first symbol of which must be a unique terminal symbol. This requirement permits DENOTE to disambiguate different syntactic phrases derived from a single nonterminal.

A few examples of abstract syntax trees for $\mathcal{L}$ should clarify these implementation requirements. The terminal symbol **PLUS** is represented by the atom `PLUS`. The binary digit 0 is represented by the atom `0`. The binary number 101 is represented by the list `(bdigits (1 0 1))`. The syntactic phrase 1001+11 is represented by the list `(plus (bdigits (1 0 0 1)) (bdigits (1 1)))`.

4. DENOTE LANGUAGE (DL)

This section describes in detail the syntax of the DENOTE language (DL) for defining denotational semantic equations and auxiliary functions. This preliminary version of DL is syntactically sir..iar to Lisp (Ref. 13), consisting of atomic terms and s-expressions (parenthesized lists). The following groups of language constructs are defined:

- the atomic terms (constants and variables)
- the built-in operators and functions
- a construct for defining denotational semantic equations (**`defsemeq`**), and additional constructs for specifying the syntactic arguments of semantic functions (**`synclause`**) and for preventing the currying of arguments (**`args`**)
- a construct for defining auxiliary functions (**`defsemfn`**)
- the control structure constructs **`if`**, **`elseif`**, and notations for applying semantic functions, auxiliary functions, and continuations (**`applycont`**) to arguments
- the binding constructs **`let`**, **`where`**, **`whererec`**, and **`lambda`**
- the grouping constructs **`lst`**, **`tuple`**, and **`synclause`** for binding new variables

A full list of the reserved words of DL is shown in the index at the back of this report.

In the sections to follow, tables are used to show groups of syntactic objects. Each table has three columns, labeled Denotations, DL, and Common Lisp, respectively. The middle column shows the DL syntax of a reserved word or construct. The first column shows its denotational representation as displayed by the Scribe text formatter. The third column shows the fragment of Common Lisp code that implements the object. The reserved words of DL are displayed in a bold, fixed-width font, as are all Common Lisp constructs. DL is case-insensitive.

4.1. Atomic Terms

Constants and variables comprise the atomic terms of DL.

4.1.1. Constants

Table 4-1 shows the syntax for the constants of DL. These constants include the empty derivation **`epsilon`**, the empty set **`emptyset`**, the Booleans **`tt`** and **`ff`**, the numerals, atomic identifier constants, and strings.

4.1.2. Variables

Table 4-2 shows the syntax for variables of DL. All identifiers that are not reserved words of DL can be used as variables, except for **`t`**, which for historical reasons is reserved to denote a variable named TSE.[†]

[†]TSE is an acronym for Tree Structured Environment.

Denotation	DL	Common Lisp
ε	`epsilon`	`nil`
∅	`emptyset`	`nil`
tt	`tt`	`t`
ff	`ff`	`nil`
...,-1,0,1,...	`...,-1,0,1,...`	`...,-1,0,1,...`
bold-id	`'bold-id`	`'bold-id`
bold-id	`(quote bold-id)`	`'bold-id`
bold-id	`(bold bold-id)`	`'bold-id`
BOLD-ID	`(ubold bold-id)`	`'bold-id`
"xxxxx"	`(string "xxxxx")`	`"xxxxx"`

Table 4-1: Constants of DL

Denotation	DL	Common Lisp
t	`t`	`tse`
identifier	`identifier`	`identifier`
x_1	`(scriptarg (x 1))`	`x1`
x^+	`(scriptarg (x +))`	`x+`
x^*	`(scriptarg (x *))`	`x*`
x^+_2	`(scriptarg (x + 2))`	`x+2`
x^*_3	`(scriptarg (x * 3))`	`x*3`
$[[s_4]]$	`(synarg (s 4))`	`s4`
$[[s^*]]$	`(synarg (s *))`	`s*`
$[[s^+]]$	`(synarg (s +))`	`s+`
$[[s^*_5]]$	`(synarg (s * 5))`	`s*5`
$[[s^+_6]]$	`(synarg (s + 6))`	`s+6`
$[[s]]$	`(synarg s)`	`s`
[[**ID**]]	`(synarg 'id)`	`'id`
[[**ID**]]	`(synarg (quote id))`	`'id`

Table 4-2: Variables of DL

In addition, DL has two notations for introducing subscripted variable names into denotational semantic descriptions. These notations are headed by the identifiers **scriptarg** and **synarg**. The **scriptarg** notation can be applied to any variable, whereas only syntactic variables can be contained within synarg. When a semantic function is applied to arguments (the first of which must be syntactic), the first argument must be enclosed in the synarg form, to distinguish it from the remainder of the (nonsyntactic) arguments.† This first argument may be the empty sequence epsilon, a (quoted) terminal symbol, a pseudo-terminal symbol (or sequence thereof), or a nonterminal symbol (or sequence thereof).

Denotation	DL		Common Lisp
$\neg x$	(not x)		(not x)
$x \vee y$	(or x y)		(or x y)
$x \wedge y$	(and x y)		(and x y)
$x = y$	(eq x y)	(= x y)	(equal x y)
$x \neq y$	(neq x y)	(~= x y)	(not (equal x y))
$x < y$	(lt x y)	(< x y)	(< x y)
$x \leq y$	(le x y)	(<= x y)	(<= x y)
$x > y$	(gt x y)	(> x y)	(> x y)
$x \geq y$	(ge x y)	(>= x y)	(>= x y)
$-x$	(minus x)	(- x)	(- x)
$x + y$	(plus x y)	(+ x y)	(+ x y)
$x - y$	(minus x y)	(- x y)	(- x y)
$x * y$	(mult x y)	(* x y)	(* x y)
x / y	(divide x y)	(/ x y)	(quotient x y)
$x \wedge y$	(expt x y)	(^ x y)	(expt x y)
$x \in y$	(member x y)		(member x y :test 'equal)
$x \cup y$	(union x y)		(union x y :test 'equal)
$x \cap y$	(intersection x y)		(intersection x y :test 'equal)

Table 4-3: Built-in Infix Operators of DL

4.2. Built-in Infix Operators and Built-in Functions

Table 4-3 shows the syntax for the built-in operators of DL that have infix denotations. The built-in operators consist of the Boolean relations negation (not), conjunction (or), and disjunction (and); the logical relations equality (eq,=) and disequality (neq,~=); the integer inequalities less than (lt,<), less than or equal (le,<=), greater than (gt,>), and greater than or equal (ge,>=); the integer operators unary negation (minus,-), addition (plus,[+]), subtraction (minus,-), multiplication (mult,*), division (divide,/), and exponentiation (expt,^); and the set operators membership (member), union (union), and intersection (intersection).

Table 4-4 shows the syntax for the built-in functions of DL, i.e., those built-in operators that have no infix denotations. These built-in functions are list head (hd), list tail (tl), list length (length), dotted-list construction (cons), list type identification (consp), list append (append), conventional list construction (list), identifier concatenation (catenate), integer maximum (max), integer minimum (min), integer

†A syntactic argument s is distinguished by the denotation [[s]].

remainder (**rem**), integer modulus (**mod**), and integer absolute value (**abs**). Adding new built-in operators and functions to DL, as the need arises, is not difficult.

Denotation	DL	Common Lisp
hd(x)	`(hd x)`	`(car x)`
tl(x)	`(tl x)`	`(cdr x)`
length(x)	`(length x)`	`(length x)`
cons(x,y)	`(cons x y)`	`(cons x y)`
consp(x)	`(consp x)`	`(consp x)`
append(x,y)	`(append x y)`	`(list x y)`
list(x_1,x_2,...)	(list x_1 x_2 ...)	`(list x1 x2 ...)`
catenate(x,y)	`(catenate x y)`	`(pack* x y)`
max(x,y)	`(max x y)`	`(max x y)`
min(x,y)	`(min x y)`	`(min x y)`
rem(x,y)	`(rem x y)`	`(rem x y)`
mod(x,y)	`(mod x y)`	`(mod x y)`
abs(x)	`(abs x)`	`(abs x)`

Table 4-4: Built-in Functions of DL

4.3. Semantic Equations

The DL expression that defines a single denotational semantic equation is a six-element list headed by the identifier **defsemeq**, whose remaining five fields contain the following:

2. the atomic name of the semantic function,
3. a unique atomic label for the semantic equation,
4. the syntactic argument of the equation,
5. a list of the arguments to the function, and
6. the body of the equation.

For example, consider the following **defsemeq** form:

```
(defsemeq
  E E4 (synclause expr ('divide (expr 1) (expr 2))) (c)
   (where (((scriptarg (c 1))
            (lambda (x)
              (where (((scriptarg (c 2))
                       (lambda (y)
                         (if (= x 0)
                             (division-error)
                             (applycont c (/ y x))))))
                  (E (synarg (expr 1)) (scriptarg (c 2)))))))
         (E (synarg (expr 2)) (scriptarg (c 1)))))
```

This form defines a semantic function named E and labeled E4, with the syntactic argument

[[**DIVIDE** $expr_1$ $expr_2$]], the semantic arguments (c), and a body that is best explained by the remainder of this report. The interior constituents of the `defsemeq` form are the topic of the remainder of this section. Semantic equation E4 is also presented in the Appendix.

4.3.1. Names and Labels

Each semantic equation defined via `defsemeq` first identifies the name of the semantic function for which an instance is being defined. Since each semantic function operates on one or more syntactic domains, several semantic equations may be required to define the actions of a particular semantic function on different syntactic phrases. Thus, DENOTE requires that a unique (atomic) label be associated with each equation. These labels may be used to refer to semantic equations in documents and are used to construct names for Common Lisp functions that implement the semantic equations. A typical methodology for constructing label names would be to append consecutive numerals to semantic function names. For example, the labels E1, E2, E3, and E4 are used to identify uniquely the four semantic equations for the semantic function E of the example language $\mathcal{L}$ (described in the Appendix).

4.3.2. Syntactic Arguments

Each semantic equation defined via `defsemeq` specifies the actions of a semantic function on a particular syntactic phrase. DENOTE requires that the syntactic argument to each semantic equation be represented by a syntactic clause, which contains both a syntactic phrase and the Nonterminal or pseudoterminal from which the phrase was derived. The DL expression defining a semantic clause is a three-element list headed by the identifier `synclause`, whose remaining two fields contain, respectively, a nonterminal or pseudoterminal symbol (or sequence thereof) for one of the syntactic phrases it derives. The syntax of both arguments to `synclause` must conform to that imposed on symbols and alternatives in abstract syntax productions.

For example, consider the following syntactic clause taken from the original `defsemeq` example:

```
(synclause expr ('DIVIDE (expr 1) (expr 2)))
```

This is the syntactic clause that represents a binary expression involving division, derived from the following two productions in the abstract syntax for $\mathcal{L}$:

expr ::= binary-op $expr_1$ $expr_2$

binary-op ::= PLUS | MINUS | MULT | DIVIDE

Below are some examples of syntactic clauses involving sequences of the nonterminal `bdigit`. To the right of each clause is the embedded phrase as it would appear as the syntactic argument in a semantic equation.

`(synclause (bdigit * 1) (bdigit (bdigit * 2)))`	[[bdigit $bdigit^*_2$]]
`(synclause (bdigit *) epsilon))`	[[ε]]
`(synclause (bdigit *) (bdigit *))`	[[$bdigit^*$]]

```
(synclause (bdigit +) (bdigit +))                [[bdigit+]]

(synclause (bdigit + 0) ((bdigit *) bdigit))     [[bdigit* bdigit]]
```

All of the above examples conform to the DL syntax, and are appropriately qualified. Note that qualification is unnecessary if the alternative (the second argument to **synclause**) is identical to the first argument, as shown in the third and fourth examples above.

4.3.3. Semantic Arguments

The semantic arguments of each semantic equation are represented by a list of identifiers; each identifier in the list represents a single argument to the function. Note that since *currying* is the default in denotational semantics, we need additional notation for grouping arguments. The notation

```
(args a b c ...)
```

groups a set of arguments together, that is, it *uncurries* them. For example, the DL notation `(a (args b c) d e)` denotes the following argument list: (a)(b,c)(d)(e). If the **args** notation is used to define the arguments to a function, the notation must be used correspondingly when the function is applied to arguments.

4.4. Auxiliary Functions

Before describing the DL constituents of the bodies of equations and functions, we must describe a form used to define auxiliary functions that may be used by semantic equations. The DL expression defining an auxiliary function is a four-element list headed by the atom **defsemfn**, whose remaining three fields contain the following:

2. the atomic name of the function,
3. a list of the names of the function's parameters (which may be grouped using **args**), and
4. the body of the function.

For example, the factorial function for non-negative integers is defined in DL as follows:

```
(defsemfn factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

4.5. Equation and Function Bodies

The bodies of semantic equations and auxiliary semantic functions, of course, comprise the majority of DL. These bodies consist of terms, some of which are special denotations.

Denotation	DL	Common Lisp
$(p \rightarrow a, b)$	`(if p a b)`	`(if p a b)`
$(p \rightarrow a,$	`(elseif p a`	`(cond (p a)`
$q \rightarrow b,$	`q b`	`(q b)`
$\ldots,$	`...`	`...`
$c)$	`c)`	`(t c))`
$\underline{SF}[[s]](x)(y)\ldots$	`(SF (synarg s) x y ...)`	`(SF s x y ...)`
$f(x_1)(x_2)\ldots$	`(f x1 x2 ...)`	`(f x1 x2 ...)`
$f(x_1,x_2,\ldots)$	`(f (args x1 x2 ...))`	`(f x1 x2 ...)`
$f(x_1)(x_2)(x_3,x_4,\ldots)$	`(f x1 x2 (args x3 x4 ...))`	`(f x1 x2 x3 x4 ...)`
$k(a)(b)(c)$	`(applycont k a b c)`	`(apply-cont k (list a b c))`
$k(a,b,c)$	`(applycont k (args a b c))`	`(apply-cont k (list a b c))`

Table 4-5: Control Structures in DL

Control Structures

Table 4-5 shows the DL syntax for the control structures used in the bodies of denotational equations and functions. The forms described are the two conditionals **`if`** and **`elseif`**, the form for semantic function application (note that the first argument to a semantic function must be a syntactic argument, enclosed by the **`synarg`** construct, as discussed previously), the form for general function application, and the form for applying continuation variables to arguments (`applycont`).

Grouping Constructs

Table 4-6 show the syntax of DL's constructs for binding variables and functions, including the **`where`** and **`let`** constructs for binding variables in various manners, the **`whererec`** form for locally binding a name to a recursive function, and the **`lambda`** forms for function definition. The **let** and **where** forms permit four distinct types of bindings: (1) binding of a variable to the empty sequence (the empty list), (2) binding of a variable to the value of an expression, (3) binding of sequences (lists and tuples) of variables to elements of lists, and (4) binding of syntactic variables in the syntactic phrase of a syntactic clause to a list of syntactic values.

Grouping Operators

Table 4-7 shows the three grouping operators for the **`let`** and **`where`** control structures, which display as parenthesized lists, and angle-bracketed lists (tuples). Note that the identifier **`lst`** is used instead of **`list`** for denotations of the form $(x_1,x_2,\ldots)$. This is because **`list`** is reserved for denotations of the form $\text{list}(x_1,x_2,\ldots)$, which represents the list constructor operator as applied to arguments.

Denotation	DL	Common Lisp
let x = ε ...	`(let (x) ...)`	`(let (x) ...)`
let x = y ...	`(let ((x y)) ...)`	`(let ((x y)) ...)`
let (x1,...,xk) = y ...	`(let (((lst x1 ... xk) y)) ...)`	`(let<k> ((x1 ... xk) y) ...)`
let <x1,...,xk> = y ...	`(let (((tuple x1 ... xk) y))...)`	`(let<k> ((x1 ... xk) y) ...)`
let [[x1,...,xk]] = y ...	`(let (((synclause s (s1 ... sk)) e)) ...)`	`(let<k> ((s1 ... sk) y) ...)`
... where x = ε	`(where (x) ...)`	`(let (x) ...)`
... where x = y	`(where ((x y)) ...)`	`(let ((x y)) ...)`

and so on. The remaining **where** *denotations follow the pattern of the* **let** *denotations:*

Denotation	DL	Common Lisp
... whererec f(a)(b)...=exp	`(whererec ((f (lambda (a b ...) exp))) ...)`	`(labels ((f (a b ...) exp)) ...)`
λx.λy.λz...body	`(lambda (x y z ...) body)`	`(function (lambda (x y z ...) body))`
λ(x,y,z,...).body	`(lambda ((args x y z ...)) body)`	`(function (lambda (x y z ...) body))`

Table 4-6: Binding Constructs of DL

Denotation	DL	Common Lisp
x = ε	`(x y ...)`	bind x to NIL, y to NIL, ...
x = y	`((u v) (w x))`	bind u to the value of v, w to x, etc.
(x1,x2,...) = y	`(((lst x1 x2 ...) y))`	binds each x_n to the nth element of y
<x1,x2,...> = y	`(((tuple x1 x2 ...) y))`	binds each x_n to the nth element of y
[[s1,s2,...]] = y	`(((synclause s (s1 s2 ...)) y))`	binds each s_n to the nth element of y

Table 4-7: The Syntax of Bindings Within **let** and **where**

5. SUMMARY

The design and implementation of the DENOTE tool was necessitated by the Computer Verification Group's problems with *ad hoc* implementations of translators for computer languages dealt with by SDVS. DENOTE now permits users to specify the formal denotational semantics of a computer language in its internal language DL, and is capable of automatically transforming a DL specification into text that may be examined for semantic correctness, as well as into a Common Lisp implementation of the specified semantics. DENOTE is currently used to define the translation semantics of Ada, VHDL, and ISPS, which permits SDVS to translate computer language constructs into the internal language of SDVS. DENOTE is not, however, restricted to defining SDVS translation semantics. It is a general-purpose tool for displaying and implementing the denotational semantics of computer languages.

The current implementation of DENOTE has been found to be adequate to the tasks presented to it, but has room for improvement. Some improvements that would extend its applicability and usability include the following:

- Improving the checking capabilities of DENOTE with respect to static semantics, e.g. type-check variables and function parameters, check that function calls have the required number of arguments.
- Implementing a friendlier user interface to DENOTE, and providing good documentation for the interface (no such documentation is provided by this report). This interface should include a facility for processing an individual semantic equation or auxiliary function, whereas the current version of DENOTE requires the processing of the entire denotational semantic definition.
- Incorporating additional denotational constructs into DL, and modifing the syntax of DL to clear up confusions or ambiguities.
- Providing an optimizer for the automatically generated Common Lisp code.
- Improving the formatting capabilities of DENOTE with respect to the Scribe and LaTEX text formatters.

A report describing DENOTE in full, including its user interface, will be forthcoming.

REFERENCES

[1] Michael J. C. Gordon.
The Denotational Description of Programming Languages: An Introduction.
Springer-Verlag, New York, 1979.

[2] *SCRIBE Introductory User's Manual*
2 edition, Unilogic Ltd., Pittsburgh, Pa, 1979.

[3] Leslie Lamport.
LaTEX: A Document Preparation System.
Addison-Wesley, Reading, Massachusetts, 1986.

[4] Guy L. Steele, Jr.
Common LISP: The Language.
Digital Press, 1984.

[5] L. Marcus.
SDVS 7 Users' Manual.
Technical Report ATR-88(3778)-5, The Aerospace Corporation, 1988.

[6] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek.
The ISPS Computer Description Language.
CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, August 1979.

[7] Bolt Beranek and Newman Inc.
C/30 Native Mode Firmware System, Programmer's Reference Manual.
Technical Report 5000, Bolt Beranek and Newman Inc., November 1983.

[8] Anthony Lake et al.
Flexible Processor Extends Design Options.
Computer Design :181-186, November 1981.

[9] J. V. Cook.
Final Report for the C/30 Microcode Verification Project.
Technical Report ATR-86(6771)-3, The Aerospace Corporation, September 1986.

[10] D. F. Martin and J. V. Cook.
Adding Ada Program Verification Capability to the State Delta Verification System (SDVS).
In *Proceedings of the 11th National Computer Security Conference.* National Bureau of Standards / National Computer Security Center, Baltimore, Md, 1988.

[11] D. F. Martin.
A Formal Description of the Incremental Translation of Core Ada into State Deltas in the SDVS Proof System.
Technical Report ATR-88(3778)-1, The Aerospace Corporation, 1988.

[12] *IEEE Standard VHDL Language Reference Manual*
IEEE, 1988.
IEEE Std. 1076-1987.

[13] John McCarthy.
LISP 1.5 Programmer's Manual.
Technical Report, M. I. T., 1962.

APPENDIX
THE LANGUAGE $\mathcal{L}$

A denotational semantics for the example language $\mathcal{L}$ (introduced in Section 3) is presented in this appendix. First the concrete and abstract syntaxes are given for $\mathcal{L}$. Next the semantics for $\mathcal{L}$ is described informally, followed by its formal semantics written in DL. Two outputs produced by DENOTE, the text-formatted version of the semantics of $\mathcal{L}$ and its Common Lisp implementation, are then presented. Finally, some examples of using the Common Lisp implementation to determine the semantics of sentences in $\mathcal{L}$ are shown.

1. Concrete Syntax for $\mathcal{L}$

The concrete syntax of $\mathcal{L}$ is shown below. As mentioned previously, each sentence in $\mathcal{L}$ represents an arbitrary arithmetic term involving the operations of addition, subtraction, multiplication, and division applied to sequences of binary digits.

```
expr    ::= sum
sum     ::= factor addop sum | factor
factor  ::= term mulop factor | term
term    ::= bnum | ( expr )
addop   ::= + | -
mulop   ::= * | /
bnum    ::= bdigit bnum | bdigit
bdigit  ::= 0 | 1
```

2. Abstract Syntax for $\mathcal{L}$

The abstract syntax of $\mathcal{L}$ is shown below. The abstract syntax productions generate the same language that concrete syntax does, while eliminating extraneous productions related to parsing, and thus is considerably more compact.

expr ::= binary-op expr_1 expr_2 | **BDIGITS** bdigit^+
binary-op ::= **PLUS** | **MINUS** | **MULT** | **DIVIDE**
bdigit ::= 0 | 1

Shown below is the Common Lisp syntax for binding the variable **`*binexpr-abstract-syntax*`** to the DL abstract syntax of $\mathcal{L}$. The abstract syntax representation is case-insensitive. We a follow a convention when displaying abstract syntax productions of presenting terminal symbols in upper case, and all other symbols in lower case.

```
(defvar *binexpr-abstract-syntax*
        '((expr (binary-op (expr 1) (expr 2))
                ('bdigits (bdigit +)))
          (binary-op 'plus 'minus 'mult 'divide)
          (bdigit '0 '1)))
```

3. Informal Semantics for $\mathcal{L}$

Informally, we choose a semantics for $\mathcal{L}$ that assigns integers to sentences in $\mathcal{L}$, except for those that involve division by zero, to which we assign an error string. In this semantics, sequences of binary digits represent non-negative binary numbers.

4. Formal Semantics for $\mathcal{L}$ in DL

The DL representation of the formal semantics for $\mathcal{L}$ is shown below. This is a continuation-style denotational semantics that maps illegal expressions (those involving division by 0) into an error string and legal expressions to their integer results. The values of three variables are defined. The variable **`*binexpr-semfns*`** is bound to an association list of semantic function names, where the tail of each sublist contains the syntactic domains (specified by the nonterminals of the abstract syntax) to which the semantic function may be applied. The variable **`*binexpr-fns*`** is bound to an association list of the auxiliary functions. Finally, the variable **`*binexpr-semantics*`** is bound to the DL semantics for $\mathcal{L}$.

```
(defvar *binexpr-semfns*
  '((A expr)
    (B expr)
    (E (bdigit *))))

(defvar *binexpr-fns*
   '((division-error)))

(defvar *binexpr-semantics*
  '((defsemeq
      A A1 (synclause expr expr) ()
      (E (synarg (expr)) (lambda (x) x)))

    (defsemeq
      E E1 (synclause expr ('plus (expr 1) (expr 2))) (c)
      (E (synarg (expr 1))
      (lambda (x) (E (synarg (expr 2)) (lambda (y) (applycont c (+ x y)))))))

    (defsemeq
      E E2 (synclause expr ('minus (expr 1) (expr 2))) (c)
      (let (((scriptarg (c 1))
             (lambda (x)
               (E (synarg (expr 2)) (lambda (y) (applycont c (- x y)))))))
           (E (synarg (expr 1)) (scriptarg (c 1)))))

    (defsemeq
      E E3 (synclause expr ('mult (expr 1) (expr 2))) (c)
      (where (((scriptarg (c 1))
               (lambda (x) (E (synarg (expr 2))
                                  (lambda (y) (applycont c (* x y)))))))
             (E (synarg (expr 1)) (scriptarg (c 1)))))
(defsemeq
  E E4 (synclause expr ('divide (expr 1) (expr 2))) (c)
  (where (((scriptarg (c 1))
           (lambda (x)
              (where (((scriptarg (c 2))
                       (lambda (y)
                          (if (= x 0)
                              (division-error)
                              (applycont c (/ y x))))))
                     (E (synarg (expr 1)) (scriptarg (c 2)))))))
         (E (synarg (expr 2)) (scriptarg (c 1)))))


(defsemfn division-error () (string "division by zero"))

(defsemeq
  E E5 (synclause expr ('bdigits (bdigit +))) (c)
  (applycont c (B (synarg (bdigit +)) 0)))

(defsemeq
  B B1 (synclause (bdigit * 0) (bdigit (bdigit *))) (n)
  (B (synarg (bdigit *)) (+ (* n 2) bdigit)))

(defsemeq
    B B2 (synclause (bdigit * 0) epsilon) (n)
    n))
))
```

The DL semantics for $\mathcal{L}$ is explained in the next section, where the text-formatted version of the equations are displayed.

5. Scribe Formatted Denotational Semantics for $\mathcal{L}$

The Scribe-formatted text displaying the abstract syntax, semantic equations, and auxiliary functions defining the denotational semantics for $\mathcal{L}$ are show below. This text was automatically generated by DENOTE from the semantic definition of $\mathcal{L}$ written in DL.

expr ::= binary-op expr$_1$ expr$_2$ | **BDIGITS** bdigit$^+$
binary-op ::= **PLUS** | **MINUS** | **MULT** | **DIVIDE**
bdigit ::= **0** | **1**

(A1) $\underline{\mathbf{A}}[\![\text{expr}]\!] = \underline{\mathbf{E}}[\![\text{expr}]\!](\lambda x.x)$

(E1) $\underline{\mathbf{E}}[\![\textbf{PLUS}\ \text{expr}_1\ \text{expr}_2]\!](c) = \underline{\mathbf{E}}[\![\text{expr}_1]\!](\lambda x.\underline{\mathbf{E}}[\![\text{expr}_2]\!](\lambda y.c(x+y)))$

(E2) $\underline{\mathbf{E}}[\![\textbf{MINUS}\ \text{expr}_1\ \text{expr}_2]\!](c)$
  $= \textbf{let}\ c_1 = \lambda x.\underline{\mathbf{E}}[\![\text{expr}_2]\!](\lambda y.c(x-y))\ \textbf{in}$
    $\underline{\mathbf{E}}[\![\text{expr}_1]\!](c_1)$

(E3) $\underline{\mathbf{E}}[\![\textbf{MULT}\ \text{expr}_1\ \text{expr}_2]\!](c)$
  $= \underline{\mathbf{E}}[\![\text{expr}_1]\!](c_1)$
    $\textbf{where}\ c_1 = \lambda x.\underline{\mathbf{E}}[\![\text{expr}_2]\!](\lambda y.c(x \times y))$

(E4) $\underline{E}[\![$**DIVIDE** $expr_1\ expr_2]\!](c)$
$= \underline{E}[\![expr_2]\!](c_1)$
where $c_1 = \lambda x.\underline{E}[\![expr_1]\!](c_2)$
where $c_2 = \lambda y.(x = 0 \rightarrow \text{division-error}(), c(y/x))'$

division-error() = "division by zero"

(E5) $\underline{E}[\![$**BDIGITS** $bdigit^+]\!](c) = c(\underline{B}[\![bdigit^+]\!](0))$

(B1) $\underline{B}[\![bdigit\ bdigit^*]\!](n) = \underline{B}[\![bdigit^*]\!](n\times2+bdigit)$

(B2) $\underline{B}[\![\varepsilon]\!](n) = n$

The semantic function $\underline{A}$ computes the semantics of expressions, using the semantic function $\underline{E}$ to compute the semantics of subexpressions. The semantic function $\underline{B}$ computes the semantics of sequences of binary digits. $\underline{A}$ requires only one argument, the syntactic phrase (representing an arbitrary expression) whose semantics is being computed. $\underline{E}$ requires two arguments, a syntactic phrase that represents an expression and a continuation c. The continuation c is a function that generally maps integers to integers, but sometimes returns an error. If an error (division by zero) occurs, the continuation mechanism is bypassed and an error string is produced by the auxiliary function **division-error**. $\underline{B}$ takes two arguments, a (possibly empty) list of binary digits and a non-negative integer, and is used to compute the semantics of a sequence of binary digits interpreted as a non-negative binary number. Note that in this semantics we make no distinction between the denotation of a binary digit and its numeric value. Typically, one postulates the existence of a semantic function $\underline{N}$ that maps denotations of numerals to their numeric values.

6. Common Lisp Implementation of Semantics for $\mathcal{L}$

A primary output of DENOTE is a Common Lisp implementation of the specified semantics. The contents of the file generated by DENOTE for the DL semantics of $\mathcal{L}$ is shown below. The function named **binexpr-a1** is the entry point for the implementation of the semantics of abstract syntax trees of $\mathcal{L}$.

```
;;; -*- Mode: LISP; package: USER; syntax: Common-Lisp; base: 10 -*-

(defmacro defrecord (name &body body)
  '(defstruct (,name (:type list))
     .,body))

(defrecord binexpras-expr-bdigits bdigits bdigit+)

(defrecord binexpras-expr-divide divide expr1 expr2)

(defrecord binexpras-expr-mult mult expr1 expr2)

(defrecord binexpras-expr-minus minus expr1 expr2)

(defrecord binexpras-expr-plus plus expr1 expr2)

(defun binexpr-e-expr (expr c)
  (case (first expr)
    (plus (binexpr-e1 expr c))
    (minus (binexpr-e2 expr c))
    (mult (binexpr-e3 expr c))
    (divide (binexpr-e4 expr c))
    (bdigits (binexpr-e5 expr c))
    (otherwise (error "Illegal phrase of type ~a: ~a" 'expr expr))))

(defun binexpr-a1 (expr)
  (binexpr-e-expr expr #'(lambda (x) x)))

(defun binexpr-e1 (expr c)
  (let ((expr1 (binexpras-expr-plus-expr1 expr))
        (expr2 (binexpras-expr-plus-expr2 expr)))
    (binexpr-e-expr expr1
     #'(lambda (x)
         (binexpr-e-expr
           expr2 #'(lambda (y)
                     (binexprtr-apply-cont c (list (+ x y)))))))))

(defun binexpr-e2 (expr c)
  (let ((expr1 (binexpras-expr-minus-expr1 expr))
        (expr2 (binexpras-expr-minus-expr2 expr)))
    (let ((c1
           #'(lambda (x)
                (binexpr-e-expr
                  expr2 #'(lambda (y)
                            (binexprtr-apply-cont c (list (- x y)))))
                )))
      (binexpr-e-expr expr1 c1))))

(defun binexpr-e3 (expr c)
  (let ((expr1 (binexpras-expr-mult-expr1 expr))
        (expr2 (binexpras-expr-mult-expr2 expr)))
    (let ((c1
           #'(lambda (x)
                (binexpr-e-expr
                  expr2 #'(lambda (y)
                            (binexprtr-apply-cont c (list (* x y)))))
                )))
      (binexpr-e-expr expr1 c1))))

(defun binexpr-e4 (expr c)
  (let ((expr1 (binexpras-expr-divide-expr1 expr))
        (expr2 (binexpras-expr-divide-expr2 expr)))
    (let ((c1
           #'(lambda (x)
                (let ((c2
                       #'(lambda (y)
                           (if (equal x 0)
                               (binexpr-division-error)
                               (binexprtr-apply-cont
                                 c (list (truncate (/ y x))))))))
                  (binexpr-e-expr expr1 c2)))))
      (binexpr-e-expr expr2 c1))))

(defun binexpr-division-error ()
  "division by zero")
```

```
(defun binexpr-e5 (expr c)
  (let ((bdigit+ (binexpras-expr-bdigits-bdigit+ expr)))
    (binexprtr-apply-cont c (list (binexpr-b1 bdigit+ 0)))))

(defun binexpr-b1 (bdigit*0 n)
  (let ((bdigit (car bdigit*0))
        (bdigit* (cdr bdigit*0)))
    (if bdigit*
        (binexpr-b1 bdigit* (+ (* n 2) bdigit))
        (binexpr-b2 nil (+ (* n 2) bdigit)))))

(defun binexpr-b2 (epsilon n)
  (declare (ignore epsilon))
  n)

(defun binexprtr-apply-cont (cont args)
  (apply cont args))
```

7. Example Semantics for Sentences of $\mathcal{L}$

Compiling and loading the Common Lisp file whose contents were shown in the previous section gives one access to the function `binexpr-a1`, which determines the semantics of sentences of $\mathcal{L}$. Examples of the semantics of abstract syntax trees generated from sentences of $\mathcal{L}$ are shown below.

```
command: (binexpr-a1 '(bdigits (1)))

1

command: (binexpr-a1 '(bdigits (1 0 1)))

5

command: (binexpr-a1 '(bdigits (0 0 0)))

0

command: (binexpr-a1 '(plus (bdigits (1 0 1)) (bdigits (1))))

6

command: (binexpr-a1 '(minus  (bdigits (1)) (bdigits (1 0 1))))

-4

command: (binexpr-a1 '(mult (bdigits (0)) (bdigits (1 0 1))))

0

command: (binexpr-a1 '(divide  (bdigits (1 0 1)) (bdigits (0))))

"division by zero"

command: (binexpr-a1 '(divide (bdigits (1)) (bdigits (1 0 1))))

0
```

```
command: (binexpr-a1 '(divide (bdigits (1 1 1)) (bdigits (1 0 1))))

1

command: (binexpr-a1 '(plus (minus (bdigits (1 1 1)) (bdigits (1 0 1)))
                            (mult (divide (bdigits (1 1 1)) (bdigits (1 0)))
                                  (bdigits (1 1)))))

11

command: (binexpr-a1 '(plus (minus (bdigits (1 1 1)) (bdigits (1 0 1)))
                            (mult (divide (bdigits (1 1 1)) (bdigits (0 0)))
                                  (bdigits (1 1)))))

"division by zero"
```

INDEX